Einführung in moofeKIT

1 Ein einführendes Beispiel

Die wichtigsten Funktionen von moofeKIT sollen anhand eines numerischen Beispiels erläutert werden. Dazu betrachten wir die in Abbildung 1 dargestellte ebene Cooks-Membran, welche auch in der 12. Übung der Lehrveranstaltung Grundlagen Finite Elemente untersucht wird. Als Materialparameter seien der Elastizitätsmodul $E = 100$ und die Querkontraktionszahl $\nu = 0.2$ gegeben.

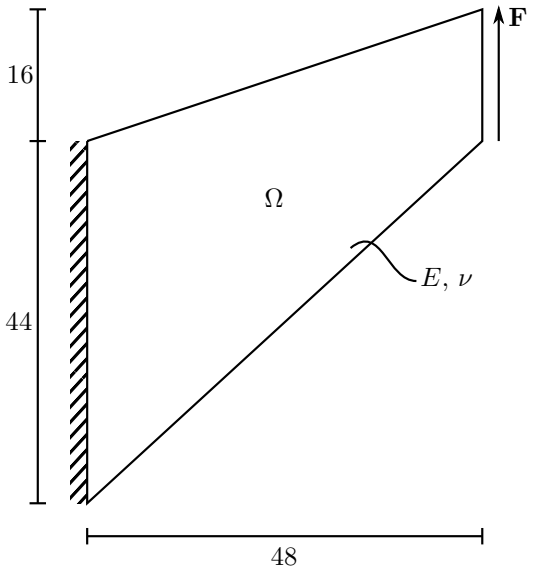

Abbildung 1: grafische Darstellung der Cooks-Membran

Dieses numerische Beispiel ist bereits in moofeKIT implementiert und wird in den nachfolgenden Abschnitten detailliert erklärt. Im Gegensatz zum Beispiel aus der Übung wird die Last $\mathbf{F}$ in unserem Fall jedoch nicht von Anfang an voll aufgebracht, sondern schrittweise über insgesamt fünf Zeitschritte hinweg gesteigert (inkrementelle Laststeigerung).

1.1 Theoretische Grundlagen: Von der schwachen Form zum Gleichungssystem

Zur Rekapitulation sollen in diesem Abschnitt die wichtigsten Gleichungen wiederholt werden. Wir gehen von einem statischen Problem aus, weshalb die schwache Form

$$\int_\Omega \nabla \mathbf{v} : \boldsymbol{\sigma} \, \mathrm{d}V = \int_\Omega \mathbf{v} \cdot \mathbf{b} \, \mathrm{d}V + \int_{\partial\Omega_\sigma} \mathbf{v} \cdot \bar{\mathbf{t}} \, \mathrm{d}A$$

unter Verwendung der Randbedingungen aus den lokalen statischen Feldgleichungen hergeleitet werden kann. Dabei werden mit $\mathbf{v}$ die vektorwertigen Testfunktionen und mit Ω das gegebene

Gebiet bezeichnet. Mit dem Hookeschen Gesetz in Voigtscher Notation $\boldsymbol{\sigma}^{\mathrm{v}} = \mathbf{D}\boldsymbol{\varepsilon}^{\mathrm{v}}(\mathbf{u})$ kann die schwache Form alternativ auch durch

$$\int_\Omega \boldsymbol{\varepsilon}^{\mathrm{v}}(\mathbf{v})^{\mathrm{T}} \mathbf{D} \boldsymbol{\varepsilon}^{\mathrm{v}}(\mathbf{u}) \, \mathrm{d}V = \int_\Omega \mathbf{v} \cdot \mathbf{b} \, \mathrm{d}V + \int_{\partial\Omega_\sigma} \mathbf{v} \cdot \bar{\mathbf{t}} \, \mathrm{d}A$$

dargestellt werden. In diesem Beispiel wollen wir für die FE-Approximation das biquadratische 9-Knoten Element verwenden. Auf Elementebene verwenden wir also jeweils 9 Ansatzfunktionen $(N_I(\boldsymbol{\xi}))$ um die Lösungsfunktionen $\mathbf{u}^{\mathrm{h},e}(\boldsymbol{\xi}) = \sum_{I=1}^{9} N_I(\boldsymbol{\xi})\mathbf{u}_I$ und die Testfunktionen $\mathbf{v}^{\mathrm{h},e}(\boldsymbol{\xi}) = \sum_{I=1}^{9} N_I(\boldsymbol{\xi})\mathbf{v}_I$ zu approximieren. Jeder Knoten besitzt dabei zwei Verschiebungs-Freiheitsgrade, d.h. $\mathbf{u}_I = [u_{xI},\ u_{yI}]^{\mathrm{T}}$. Die diskreten Verzerrungen lassen sich durch die Einführung der sogenannten B-Matrix kompakt darstellen.

$$\left(\boldsymbol{\varepsilon}^{\mathrm{v}}(\mathbf{u})\right)^{\mathrm{h},e} = \begin{bmatrix} u^{\mathrm{h},e}_{x,x} \\ u^{\mathrm{h},e}_{y,y} \\ u^{\mathrm{h},e}_{x,y} + u^{\mathrm{h},e}_{y,x} \end{bmatrix} = \begin{bmatrix} \sum_{I=1}^{9} N_{I,x}(\boldsymbol{\xi}) u_{xI} \\ \sum_{I=1}^{9} N_{I,y}(\boldsymbol{\xi}) u_{yI} \\ \sum_{I=1}^{9} \left(N_{I,y}(\boldsymbol{\xi}) u_{xI} + N_{I,x}(\boldsymbol{\xi}) u_{yI} \right) \end{bmatrix} = \sum_{I=1}^{9} \underbrace{\begin{bmatrix} N_{I,x} & 0 \\ 0 & N_{I,y} \\ N_{I,y} & N_{I,x} \end{bmatrix}}_{\mathbf{B}_I} \underbrace{\begin{bmatrix} u_{xI} \\ u_{yI} \end{bmatrix}}_{\mathbf{u}_I}$$

Einsetzen der diskreten Verzerrungen in die schwache Form führt auf

$$\int_{\Omega^e} \sum_{I=1}^{9} \mathbf{v}_I^{\mathrm{T}} \mathbf{B}_I^{\mathrm{T}} \mathbf{D} \sum_{J=1}^{9} \mathbf{B}_J \mathbf{u}_J \, \mathrm{d}V = \int_{\Omega^e} \sum_{I=1}^{9} N_I \mathbf{v}_I \mathbf{b} \, \mathrm{d}V + \int_{\partial\Omega^e_\sigma} \sum_{I=1}^{9} N_I \mathbf{v}_I \bar{\mathbf{t}} \, \mathrm{d}A$$

$$\Leftrightarrow \quad \mathbf{v}_I^{\mathrm{T}} \underbrace{\sum_{I=1}^{9} \sum_{J=1}^{9} \int_{\Omega^e} \mathbf{B}_I^{\mathrm{T}} \mathbf{D} \mathbf{B}_J \, \mathrm{d}V}_{\mathbf{K}^e} \mathbf{u}_J = \mathbf{v}_I^{\mathrm{T}} \underbrace{\sum_{I=1}^{9} \left(\int_{\Omega^e} N_I \mathbf{b} \, \mathrm{d}V + \int_{\partial\Omega^e_\sigma} N_I \bar{\mathbf{t}} \, \mathrm{d}A \right)}_{\mathbf{F}^e}$$

Nach der Assemblierung der Elementbeiträge ergibt sich das zu lösende globale Gleichungssystem

$$\mathbf{K}\mathbf{q} = \mathbf{F}. \tag{1}$$

1.2 Theoretische Grundlagen: Das Newton-Raphson-Verfahren

Das Newton-Raphson-Verfahren ist ein Approximationsalgorithmus zur numerischen Lösung von nichtlinearen Gleichungen und Gleichungssystemen. Mithilfe dieses Verfahrens können Nullstellen einer stetig differenzierbaren Funktion berechnet werden, wenn direktes Auflösen nach der gesuchten Größe nicht möglich ist.

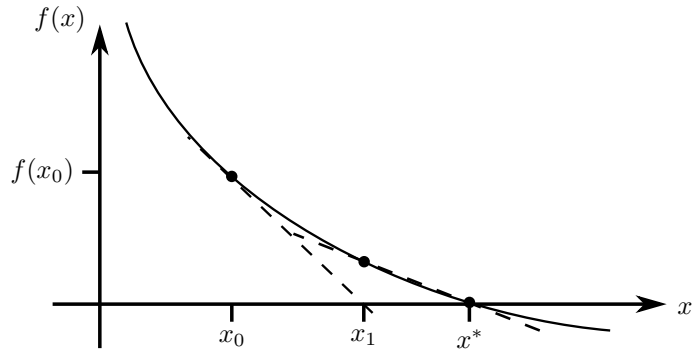

Abbildung 2: schematische Darstellung des Newton-Raphson-Verfahrens

Wir betrachten zunächst Funktionen $f(x) = 0$, die nur von einer Variablen abhängig sind. Für diese lautet die Iterationsvorschrift

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \quad i = 0, \ldots, n, \tag{2}$$

dabei wird $f(x)$ auch als Residuum (R) und $f'(x)$ als Tangente (K_T) bezeichnet.

Wie in Abbildung 2 dargestellt, ist x_0 der Startwert der Iteration. Der aus der Formel berechnete Wert x_1 wird dann neuer Startwert. Die Berechnung wird solange wiederholt bis eine Abbruchbedingung erfüllt ist. In moofeKIT wird das Verfahren abgebrochen, wenn die Norm des Residuums unterhalb eines vorgegebenen Toleranzwertes ε liegt. Übertragen auf die schematische Darstellung in Abbildung 2 ist die Nullstelle x^* der Funktion also gefunden, wenn $||\mathrm{R}|| = ||f(x^*)|| < \varepsilon$.

Für Gleichung (1) aus dem vorherigen Abschnitt wird im Folgenden das Newton-Raphson-Verfahren angewendet. Dazu muss die Gleichung so umgestellt werden, dass die rechte Seite verschwindet. Es ergibt sich für das Residuum

$$\mathbf{R} = \mathbf{f}(\mathbf{q}) = \mathbf{K}\mathbf{q} - \mathbf{F}$$

und für die Tangente

$$\mathbf{K}_\mathrm{T} = \frac{\partial \mathbf{R}}{\partial \mathbf{q}} = \mathbf{K}.$$

Analog zu (1.2) lautet die Iterationsvorschrift für dieses von mehreren Variablen abhängige Nullstellenproblem

$$\mathbf{q}_{i+1} = \mathbf{q}_i - \mathbf{K}_{\mathrm{T}_i}^{-1}\mathbf{R}_i \ , \quad i = 0, \ldots, n,$$

wobei durch $(\bullet)^{-1}$ die Inverse einer Matrix gekennzeichnet ist.
Wir können also das Newton-Raphson-Verfahren zur Lösung unseres globalen FE-Gleichungssystems heranziehen. Da wir ein lineares Gleichungssystem vorliegen haben, wäre es auch denkbar andere Verfahren, z.B. den Gauß-Algorithmus, zu verwenden. Da moofeKIT allerdings oftmals dazu verwendet wird, nichtlineare Gleichungssysteme zu lösen, wird zur Gleichungslösung standardmäßig immer das Newton-Raphson-Verfahren genutzt.

1.3 Das Startskript

In diesem Abschnitt wird der Aufbau des Startskripts `cooksMembrane2DIntroduction.m` (Pfad: `scripts/pre`) für unser einführendes Beispiel Schritt für Schritt erklärt. Parameter, die im Startskript angepasst werden können/müssen, sind direkt im MATLAB-Code gekennzeichnet.

Der moofeKIT-Code ist objektorientiert programmiert. Das bedeutet, dass z.B. im Startskript viele Simulationsparameter als Eigenschaften von Objekten angegeben werden. Objekte sind Abbilder von Klassen, in denen zugehörige Eigenschaften und Methoden definiert sind (genauere Informationen siehe Abschnitt 3.1). Im Startskript wird in Zeile 5 zunächst die Klasse `setupClass` aufgerufen, von der ein Objekt, das `setupObject`, abgeleitet wird. Dieses Objekt, auf das von fast überall im Code zugegriffen werden kann, beinhaltet wichtige allgemeine Einstellungen zur durchzuführenden Simulation. In Zeile 6 wird durch den Befehl `setupObject.totalTimeSteps = 5;` beispielsweise festgelegt, dass wir unser Problem in insgesamt fünf Zeitschritten rechnen möchten.

In Zeile 10 wird von der Klasse `dofClass` das Objekt `dofObject` abgeleitet. In diesem Objekt werden alle Informationen über das konkrete Finite-Elemente Problem gesammelt. Dazu definieren wir in Zeile 13 mit `solidObject = solidClass(dofObject);` zunächst, dass wir einen deformierbaren Körper der Klasse `solidClass` verwenden wollen. Durch diesen Befehl wird innerhalb des `dofObject` ein Eintrag zu einer Liste (Eigenschaft listContinuumObjects) hinzugefügt, die das neu angelegte Objekt `solidObject` enthält.

Für das `solidObject` müssen verschiedene Eigenschaften angegeben werden. Unter anderem ist darin das Materialgesetz hinterlegt, welches in unserem Fall das Hookesche Gesetz mit allen zugehörigen Parametern ist. Außerdem wird ein entsprechendes Netz mit Gaußpunkten erstellt und die Formfunktionen definiert. Die Formfunktionen sowie deren Ableitungen sind in dem `solidObject.shapeFunctionObject` abgespeichert.

Neben der Definition des betrachteten Körpers über das `solidObject`, müssen wir noch den Dirichlet-Rand und den Neumann-Rand festlegen. Der Dirichlet-Rand wird definiert, indem in Zeile 25 die `dirichletClass` aufgerufen wird und das `dirichletObject` erstellt wird. Der Neumann-Rand wird über das `neumannObject` definiert, welches in Zeile 32 aus der `neumannClass` erstellt wird. Dieses Objekt beinhaltet die Formfunktionen, Gaußpunkte und Freiheitsgrade für den Neumann-Rand sowie die angreifende Kraft und deren zeitlichen Verlauf.

Danach wird die Funktion `runNewton.m` in Zeile 42 aufgerufen. In dieser Funktion findet die eigentliche Berechnung sowie die grafische Ausgabe statt. Für mehr Details sei an dieser Stelle auf den nächsten Abschnitt verwiesen. Zu guter Letzt wird im Startskript im Zuge des Postprocessings die Verschiebung des rechten oberen Knotens ausgegeben.

1.4 runNewton.m und callElements.m

Zu Beginn der Funktion `runNewton.m` (Pfad: `scripts/solver`) werden die Variablen der Verschiebung $\mathbf{q}_n$ und der Geschwindigkeit $\mathbf{v}_n$ initialisiert. Danach beginn die Zeitschleife, implementiert als for-Schleife, mit der vorher festgelegten Anzahl an Zeitschritten. Daraufhin beginnt das Newton-Verfahren, implementiert als while-Schleife. Zunächst werden in der Schleife die Variablen geupdated und danach das Residuum und die Tangente gesucht, um einen Iterationsschritt durchzuführen. Dazu wird die Funktion `callElements.m` (Pfad: `commonFunctions`) aufgerufen, in welcher sich die Elementschleife, implementiert als parallelisierte for-Schleife (parfor), befindet. Innerhalb dieser Schleife wird die passende Elementroutine mithilfe der MATLAB-Funktion `feval` (genaueres siehe Abschnitt 4.3.13) aufgerufen. Die Elementroutine berechnet dann für jedes Element das Residuum und die Tangente.

Nach vollständigem Durchlaufen der Elementschleife werden die Elementtangenten (`kData`) und -residuen (`rData`) gespeichert und an die Funktion `runNewton.m` übergeben. Dort werden sie assembliert und es kann nun der erste Schritt des Newton-Raphson-Verfahrens berechnet werden. Wenn die Norm des Residuums oberhalb der vorgegebenen Toleranz ist, wird mit dem neu berechneten Verschiebungsvektor die Elementroutine nochmals durchlaufen. Insgesamt werden so viele Iterationsschritte durchgeführt, bis das Ergebnis unterhalb der Toleranz liegt. Bei uns braucht es nur eine Iteration, da das Problem linear ist. Am Ende der Funktion `runNewton.m` wird der Graph geplotted.

1.5 Die Elementroutine

Wie bereits erwähnt, wird über die Funktion `callElements.m` automatisiert die passende Elementroutine aufgerufen. Die Vorgaben, welche für die Auswahl der passenden Elementroutine notwendig sind, werden innerhalb des Startskript angegeben. Mit den Angaben aus unserem Startskript landen wir innerhalb der Elementroutine `displacementHooke2DEndpoint.m` (Pfad: `continuumClasses/@solidClass`). Neben dieser Elementroutine sind innerhalb von moofeKIT auch viele weitere mit unterschiedlichen Dimensionen, Materialgesetzen, etc. implementiert.

Innerhalb von `displacementHooke2DEndpoint.m` wird zunächst der Elastizitätstensor $\mathbf{D}$ berechnet. Danach beginnt die Gaußschleife, innerhalb derer die B-Matrix ermittelt, sowie das Residuum und die Tangente auf über eine Gaußintegration berechnet werden.

2 Wichtige Informationen zur Arbeit mit moofeKIT

2.1 Wichtige Grundsätze

Der Forschungscode moofeKIT ist ein Programm, welches im Rahmen der Forschungsarbeit von mehreren Personen gleichzeitig verwendet und weiterentwickelt wird. Es ist daher wichtig, dass sich alle an bestimmte Grundsätze halten, um moofeKIT möglichst übersichtlich und funktionsfähig zu halten. Die wichtigsten Grundsätze sind:

- Eigene Änderungen dürfen den Code nicht so verändern, dass moofeKIT in Teilen oder im Ganzen funktionsunfähig wird.
- In moofeKIT sollte alles auf Englisch verfasst werden (Variablennamen, Funktionsnamen, Kommentare, Ordernamen, Dateinamen, ...).
- Bezeichnungen für Variablen, Funktionen, Order und Dateien sollten immer ausgeschrieben werden und so eindeutig benannt sein, dass der Code auch für andere Personen schnell nachvollziehbar ist.
- Der erste Buchstabe einer Bezeichnung sollte immer klein geschrieben werden. Ist die Bezeichnung aus mehreren Wörtern zusammengesetzt, sollten die ersten Buchstaben aller weiteren Wörter groß geschrieben werden. Beispiel: `materialParameterMu` und nicht z.B. `Mat_ParamMu`.
- Code-Dopplungen sind zu vermeiden. Vorhandene Dopplungen sollten möglichst zügig entfernt werden.
- Funktionen sollten nur eine einzige Aufgabe erfüllen. Diese sollte aus der Bezeichnung und der Beschreibung klar hervorgehen.
- Sämtliche Skripte und Funktionen sollten durch Kommentare gut nachvollziehbar dokumentiert werden.
- Neu geschriebene Skripte und Funktionen sollten mithilfe des Quellcode-Formatierers *MBeautifier* optimiert werden (siehe Abschnitt 4.1).

2.2 Git Konzept

Der große Vorteil von Git ist, dass mehrere Personen zeitgleich sowie unabhängig voneinander an einem Projekt arbeiten können und Änderungen auch über große Zeiträume hinweg nachvollziehbar sind. Für moofeKIT gibt es ein eigenes Git-Projekt, welches über mehrere sogenannte *branches* verfügt. Ein branch repräsentiert eine unabhängige Entwicklungslinie des Hauptprojekts. In moofeKIT verwenden wir dabei das Konzept der sogenannten feature branches. Das heißt, dass jede neue Funktionalität innerhalb eines eigenen themenbezogen benannten branches entwickelt wird. Das ist von Vorteil, da dadurch ungestört von allen anderen Entwicklungen an einem neuen feature gearbeitet werden kann. Ist das feature in seinem branch fertig implementiert, kann ein merge request in den masterExperimental branch gestellt werden. Wird dieser merge request angenommen, werden die Inhalte des feature branches in den masterExperimental branch aufgenommen. Der masterExperimental branch ist ein spezieller branch des Projekts, der die aktuellste aber eventuell nur eingeschränkt nutzbare Version von moofeKIT enthält.
Für den Forschungscode moofeKIT gibt es in regelmäßigen Abständen Meetings, in denen hauptsächlich über neue Ideen für den Code diskutiert wird. Außerdem findet nach jedem Meeting ein Versionsupdate statt. Das heißt, dass dann der masterExperimental branch in den master branch gemerged wird. Da der master branch immer komplett lauffähig sein sollte, müssen eventuelle Fehler im masterExperimental branch dementsprechend immer möglichst zeitnah

behoben werden. Die Ergebnisse der einzelnen Meetings sind in der Datei `changelog.md` einsehbar. Dort werden auch die wichtigsten Code-Änderungen dokumentiert.
Nützliche Git-Befehle sind in Abschnitt 4.2 gesammelt.

2.3 Ordnerstruktur

Der Forschungscode ist in mehrere Ordner unterteilt. In diesen Ordnern finden sich die Skripte, Funktionen und Klassendefinitionen, aus denen sich moofeKIT zusammensetzt. Die wichtigsten MATLAB-Dateien zum Verständnis des Codes und wo diese zu finden sind, sind in Abbildung 3 dargestellt.

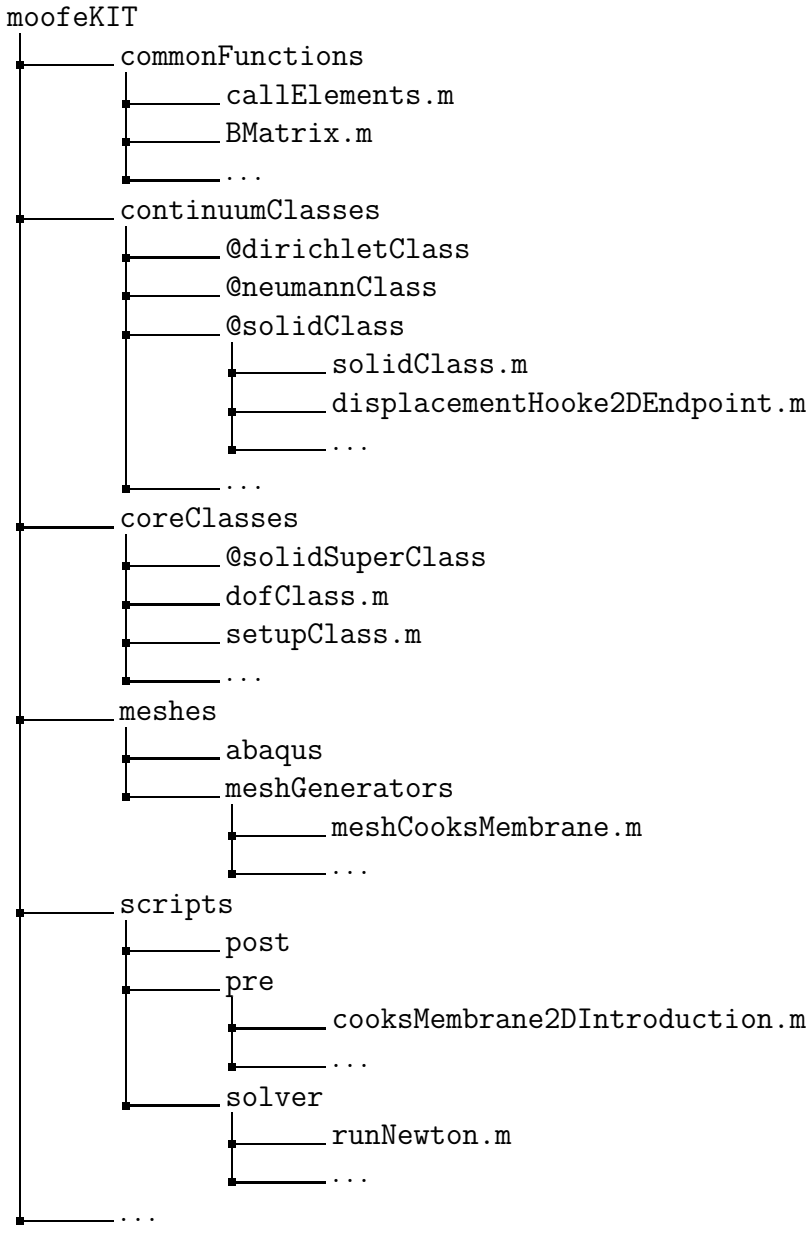

Abbildung 3: Ordnerstruktur von moofeKIT mit den wichtigsten Dateien

Der erste Ordner **commonFunctions** enthält Funktionen, die an mehreren Stellen innerhalb des Codes benötigt werden (Vermeidung von Codedopplung) oder bestimmte Funktionalitäten kompakt bündeln, z.B. *callElements.m*, *BMatrix.m*. Im zweiten Ordner **continuumClasses** sind viele Unterordner abgespeichert. Diese beinhalten je nach Name die Klasse zur Berechnung des Dirichletrandes (@dirichletClass/*dirichletClass.m*), die Klasse zur Berechnung des

Neumannrandes (@neumannClass/*neumannClass.m*) mit den Funktionen *deadLoad.m* (diese Funktion beinhaltet die Berechnung eines Neumannrandes in 3D) und die Klassen zur Berechnung von unterschiedlichen Materialgesetzen. In diesen Unterordnern sind auch jeweils die zugehörigen Elementroutinen abgespeichert, z.B. im Odner @solidClass sind sowohl die Eigenschaften und Methoden der Klasse in der Datei *solidClass.m* enthalten, als auch die Elementroutinen in Dateien wie *displacementHooke2DEndpoint.m*.

Als nächstes folgt der Ordner **coreClasses** (core = Kern, Ader). Dieser beinhaltet unter anderem die @solidSuperClass/*solidSuperClass.m*. In dieser Klasse werden Eigenschaften und Methoden definiert, die für alle continuumClasses gleich sind und daher an diese vererbt werden. Außerdem ist in dem Ordner die *dofClass.m* abgespeichert, welche Attribute wie Verschiebungen und Geschwindigkeiten der einzelnen Knoten definiert und initialisiert. Des Weiteren findet sich hier auch die *setupClass.m*, mit wichtigen allgemeinen Attributen, über die die Simulationen gesteuert werden.

Im darauffolgenden Ordner **meshes** (mesh = Netz, Gitter) gibt es den Unterordner meshGenerators, in dem alle Funktionen untergebracht sind, die ein FE-Netz erzeugen. Zum Schluss sei noch der Ordner **scripts** zu erwähnt. Darin gibt es den Unterordner pre, welcher alle Startskripte enthält, und den Unterordner solver mit der Funktion *runNewton.m*, in der die Zeit- und Newtonschleife implementiert ist. Der Unterordner post enthält Dateien für das Postprocessing (Grafikausgabe, ...).

3 Weiterführende Informationen

3.1 Objektorientierte Programmierung

Bei der objektorientierten Programmierung (OOP) wird alles in Objekten gespeichert. Beispielsweise auch Variablen, diese müssen dann nicht mehr einzeln an andere Dateien/Funktionen übergeben werden, sondern es reicht aus das passende Objekt zu übergeben. Die Objekte gehören einer bestimmten Klasse an. In dieser Klasse sind alle Attribute (Eigenschaften) und Methoden (Funktionen) definiert, die ein Objekt haben/ausführen kann. Ein Objekt ist die Instanz einer Klasse und kann mit anderen Objekten kommunizieren durch senden und empfangen von Nachrichten.

3.1.1 Klasse

Eine Klasse ist ein sogenannter Bauplan für eine bestimmte Gruppe von Objekte. Sie enthält Methoden (Funktionen) und Attribute (Eigenschaften/Konfigurationsparameter). Die Gruppierung von Objekten ist möglich durch die Zugehörigkeit zu einer bestimmten Klasse.

Wenn mehrere Objekte sich nur in wenigen Eigenschaften unterscheiden, aber der restliche Aufbau gleich ist, kann mit Vererbung gearbeitet werden. Dass bedeutet, die gleichen Eigenschaften werden in einer Basisklasse (engl. superclass) definiert. Die sich unterscheidenden Eigenschaften werden dann in Unterklassen definiert, die von der Basisklasse abgeleitet sind. So erbt die Unterklasse die Datenstruktur der Basisklasse und ergänzt sie entsprechend.

Syntax:

```
classdef Unterklasse < Basisklasse
    properties
        ...
    end
    methods
        ...
    end
end
```

3.1.2 Objekt

Wie bereits erwähnt kann auf Grundlage der Klassendefinition ein Objekt erstellt werden. Dabei beschreiben die Attribute/Eigenschaften den Zustand des Objekts. Die Methoden/Funktionen sind hingegen sämtliche Aktionen, die ein Benutzer ausführen kann. Objekte sind immer in einem wohldefinierten, selbstkontrollierten Zustand. Um zu überprüfen aus welcher Klasse ein Objekt stammt:

```
>> class(Objekt)
```

3.1.3 Attribute

Die Attribute sind die Eigenschaften eines Objekts. Wer Zugriff auf diese Eigenschaften hat, kann in der zugehörigen Klasse definiert werden:

- Get Access = private: Eigenschaft nur sichtbar für Methoden, die mit ihr arbeiten (schreibgeschützt)
- constant: keine Änderung der Eigenschaft möglich
- dependent: Eigenschaft wird nur berechnet, wenn sie angefordert wird → Get-Methode angeben, die beim Zugriff auf die Eigenschaft automatisch aufgerufen wird

Syntax:

```
properties
    ...
end
properties(SetAccess = private)
    ...
end
properties(Constant)
    ...
end
properties(Dependent = true)
    ...
end
```

3.1.4 Methoden

Methoden sind Funktionen, die in einer bestimmten Klasse definiert sind. Auf diese Funktionen können je nach Zugriffsebene unterschiedlich viele Objekte zugreifen und diese ausführen. Im Forschungscode wurden drei Zugriffsebenen verwendet:

- öffentliche (public) Methoden: dürfen von allen Klassen und deren Instanzen aufgerufen werden
- geschützte (protected) Methoden: dürfen von Klassen im selben Paket und abgeleiteten Klassen aufgerufen werden
- private (private) Methoden: können nur von anderen Methoden derselben Klasse aufgerufen werden

Syntax:

methods
  function $ordinaryMethod(obj, arg1, ...)$
    ...
  end
end

Konstruktormethoden dienen der Dateninitialisierung und -validierung eines Objekts.
Syntax:

methods
  function $obj = set.PropertyName(obj, value)$
    ...
  end
  function $value = get.PropertyName(obj)$
    ...
  end
end

3.1.5 Ziel/Vorteile

- Vererbung: Dopplungen im Forschungscode vermeiden
- Polymorphie: gleiche Nachricht löst bei unterschiedlichen Objekten entsprechend andere Aktionen aus
- Datenkapselung: keine unkontrollierte Änderung des Objekts von außen
- bessere Strukturierung des Codes

3.1.6 Weiterführende Literatur

- `https://de.wikipedia.org/wiki/Objektorientierte_Programmierung`
- `https://de.mathworks.com/company/newsletters/articles/introduction-to-object-oriente`

4 Anhang

4.1 MBeautifier

Der Quellcode-Formatierer *MBeautifier* dient dazu, selbstgeschriebenen MATLAB-Code zu verschönern. Durch die Anwendung dieses Formatierers werden bspw. Kommas und Leerzeichen hinzugefügt oder Codezeilen eingerückt. MBeautifier ist bereits in moofeKIT integriert und muss nur noch zum eigenen Editor hinzugefügt werden. Das geschieht über den Befehl
`MBeautify.createShortcut('editorpage');`
der im MATLAB- Kommandofenster eigegeben werden muss. Dadurch erscheint am rechten oberen Rand ein Button. Drückt man auf diesen Button, wird das aktuell geöffnete MATLAB-Fenster gemäß der Style-Richtlinien von MBeautifier formatiert.
In moofeKIT sollten alle neu erstellten oder bearbeiteten Dateien in dieser Weise formatiert werden, da hierdurch die Lesbarkeit des Codes verbessert wird.

4.2 Wichtige Git Befehle

4.2.1 Ersteinrichtung

Siehe Einführung GitLab

4.2.2 Einen eigenen Branch erstellen

- zum Eigenständigen und unabhängigen Arbeiten
- neuen feature branch erstellen (git branch branch)
- checkout aus eigenem branch (git checkout branch)
- add, commit und push nachdem am branch gearbeitet wurde

4.2.3 merge Request an master branch

- add, commit and push the feature branch
- switch to master branch (git checkout master)
- pull master branch to latest state (git pull)
- switch back to feature branch (git checkout branch)
- merge feature branch with master branch (git merge master)
- push feature branch to remote (git push)
- submit merge request via the webpage (recommend) and assign suitable reviewers

4.2.4 Weiterführende Literatur

- `https://git.scc.kit.edu/ifm/anleitungen/software/-/blob/master/git/simple_cheat_sheet`

4.3 Wichtige Funktionen/Befehle in MATLAB

4.3.1 assert

„throw error if condition false“
Mithilfe dieser Funktion kann eine bestimmte Bedingung überprüft werden.

$$assert(cond)$$

Wenn die Bedingung *cond* (z.B. $x < 1$) nicht eingehalten ist, wird eine Fehlermeldung ausgerufen.

4.3.2 strcmp

„compare strings“
Hier werden zwei Zeichenketten verglichen. Wenn sie identisch sind, ist der 'output' 1 (true), ansonsten 0 (false).

$$tf = strcmp(s1, s2)$$
$$tf = 1 \text{ (true)} \text{ oder } 0 \text{ (false)}$$

$s1$ und $s2$ können eine Kombination aus Zeichenketten und Vektoren sein. Es werden die einzelnen Einträge an gleicher Stelle miteinander verglichen, wenn die Zeichenketten beide mehr als einen Eintrag haben.

4.3.3 strcmpi

„compare strings (case insensitive)“
Wie vorherigen Abschnitt werden zwei Zeichenketten miteinander verglichen. Hier wird zwischen Groß- und Kleinschreibung nicht unterschieden. Zwischen Leerzeichen hingegen schon.

$$tf = strcmpi(s1, s2)$$
$$tf = 1 \text{ (true)} \text{ oder } 0 \text{ (false)}$$

4.3.4 ischar

„determine if input is character array“
Es wird überprüft, ob das „input argument“ ein „character array“ ist.

$$tf = ischar(A)$$
$$tf = 1 \text{ (true)} \text{ oder } 0 \text{ (false)}$$

4.3.5 isfield

„determine if input is structure array field“
Diese Funktion überprüft, ob ein $field$ in einem „structure array“ enthalten ist.

$$TF = isfield(S, field)$$
$$TF = 1 \text{ (true)} \text{ oder } 0 \text{ (false)}$$

Wenn $field$ der Name eines Feldes aus dem „structure array“ S ist, ist $TF = 1$. Sonst, gibt die Funktion 0 zurück.

4.3.6 isnumeric

„determine whether input is numeric array“
Es wird überprüft, ob das „input argument“ ein „array of numeric data type“ ist.

$$tf = isnumeric(A)$$
$$tf = 1 \text{ (true) oder } 0 \text{ (false)}$$

Numeric types in MATLAB: int8, int16, int32, int64, uint8, uint16, uint32, uint64, single, and double.

4.3.7 isprop

„True if property exists“
Wenn die Eigenschaft (unter $propertyName$ abgespeichert) existiert in dem Objekt, gibt die Funktion eine 1 wieder, sonst 0.

$$tf = isprop(obj, PropertyName)$$
$$tf = 1 \text{ (true) oder } 0 \text{ (false)}$$

4.3.8 vertcat

„Concatenate arrays vertically“
Mithilfe dieser Funktion können mehrere Matrizen in einer zusammengefasst untereinander geschrieben werden, sofern deren Anzahl an Spalten übereinstimmt.

$$C = vertcat(A, B)$$
$$C = vertcat(A1, A2, \ldots, An)$$

Im ersten Fall wird B unter A geschrieben, diese Funktion ist äquivalent zu $[A; B]$. Es können auch mehrere Matrizen vereint werden, sofern deren erste Dimension übereinstimmt, wie in Zeile 2 gezeigt.

4.3.9 full

„Convert sparse matrix to full storage“
Durch diese Funktion kann eine „sparse matrix“ in eine Matrix mit allen Einträgen überführt werden.

$$A = full(S)$$

Hier ist S die „sparse matrix“ und A die volle Matrix.

4.3.10 strcat

„concatenate strings horizontally“
Zeichenfolgen horizontal verketten

$$s = strcat(s1, ..., sN) \rightarrow s =' s1, ..., sN'$$

4.3.11 isa

„determine if input has specified data type“
Entspricht A einem bestimmten Datentyp?

$$tf = isa(A, \text{dataType})$$
$$tf = 1 \text{ (true)} \text{ oder } 0 \text{ (false)}$$

Wenn A ein Objekt ist, dann wird $tf = 1$, wenn „dataType“ entweder die Klasse oder Basisklasse von A.

4.3.12 isafield

„returns 1 if field is the name of a field of the structure array S“
Hiermit wird herausgefunden, ob im array S ein bestimmter Datentyp mit gleichem Namen vorhanden ist.

$$tf = isafield(S, (\text{field})$$
$$tf = 1 \text{ (true)} \text{ oder } 0 \text{ (false)}$$

4.3.13 feval

„evaluate function“
Mithilfe dieser Funktion kann das erste „input argument“ als Funktion mit den darauffolgenden „input arguments“ aufgerufen werden, um „output arguments“ zu generieren

$$[y_1, ..., y_N] = feval(fun, X_1, ..., X_M)$$

Es wird die Funktion fun mit den Parametern $X_1, .., X_M$ aufgerufen und es werden dann die Parameter $Y_1, ..., Y_N$ berechnet.

4.3.14 numel

„number of array elements“ gibt die Anzahl der Elemente eines „arrays“ wieder

$$n = numel(A)$$

4.3.15 parfor

„executes for-loop iterations on workers in a parallel pool on your multi-core computer or cluster“
der Syntax ist wie bei einer for-Schleife.

$$\begin{aligned} &parfor \quad loopVar = initVal : endVal \\ &\quad\quad statements; \\ &end \end{aligned}$$

Jedoch kann dadurch an mehreren Schleifendurchläufen gleichzeitig gerechnet werden um Zeit zu sparen. Die Berechnungen in den Schleifen dürfen nicht von den vorangegangen Durchläufen abhängig sein, da die es nicht mehr chronologisch berechnet wird. Es kann keine $parfor$-Schleife innerhalb einer anderen $parfor$-Schleife erfolgen.

4.3.16 handle

Eine „handle“ ist eine Variable, die sich auf ein bestimmtes Objekt aus der handle-Klasse bezieht. Mehrere dieser handle-Variablen können sich auf ein Objekt beziehen. Sie ermöglichen es Datenstrukturen wie etwa verknüpfte Listen zu erstellen oder mit einem großen Datensatz zu arbeiten ohne ihn zu kopieren.
Die handle-Klasse ist eine abstrakte Basisklasse, mit der keine Objekte gebildet werden können. Jedoch können in Unterklassen, die von der handle-Klasse abgeleitet sind, handle-Objekte erstellt werden. Dies wurde in dem Forschungscode gemacht.